NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

APPLICATIONS OF PROBABILISTIC COMBINERS ON LINEAR FEEDBACK SHIFT REGISTER SEQUENCES

by

Nicholas J. Sharpe

December 2016

Thesis Advisor:	Pantelimon Stănică
Second Reader:	Thor Martinsen

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	16 December 2016	Master's Thesis 06-28-2015 to 12-16-2016

4. TITLE AND SUBTITLE

APPLICATIONS OF PROBABILISTIC COMBINERS ON LINEAR FEEDBACK SHIFT REGISTER SEQUENCES

5. FUNDING NUMBERS

6. AUTHOR(S)

Nicholas J. Sharpe

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Cryptography forms the backbone of modern secure communication. Many different methods are available for encrypting and decrypting data, each with advantages and disadvantages. If communicating parties require speed of encryption more than incredibly robust security, they may use a stream cipher, which is based on generating long strings of bits with linear feedback shift registers (LFSRs), then making those strings cryptographically secure by combining them with a nonlinear Boolean function called a combiner. In this thesis, we investigate a modification to the classical combiner method by introducing a (nonsecure) probabilistic randomization to the order in which the LFSRs are input into the combiner function at each bit. We implemented two different designs for the probabilistic combiner: one that randomly ordered four LFSRs and put them into a four-variable Boolean function, and another that selected only three out of four LFSRs to use as inputs in a three-variable function. Our tests on the resulting output strings show a drastic increase in complexity, while simultaneously passing the stringent randomness tests required by the National Institute of Standards and Technology for pseudorandom numbers.

14. SUBJECT TERMS

cryptography, pseudorandom number generation, linear feedback shift register, combiner, linear complexity

15. NUMBER OF PAGES 57

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited

APPLICATIONS OF PROBABILISTIC COMBINERS ON LINEAR FEEDBACK SHIFT REGISTER SEQUENCES

Nicholas J. Sharpe
Lieutenant, United States Coast Guard
B.S., United States Coast Guard Academy, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
December 2016

Approved by: Pantelimon Stănică
Thesis Advisor

Thor Martinsen
Second Reader

Craig Rasmussen
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Cryptography forms the backbone of modern secure communication. Many different methods are available for encrypting and decrypting data, each with advantages and disadvantages. If communicating parties require speed of encryption more than incredibly robust security, they may use a stream cipher, which is based on generating long strings of bits with linear feedback shift registers (LFSRs), then making those strings cryptographically secure by combining them with a nonlinear Boolean function called a combiner. In this thesis, we investigate a modification to the classical combiner method by introducing a (nonsecure) probabilistic randomization to the order in which the LFSRs are input into the combiner function at each bit. We implemented two different designs for the probabilistic combiner: one that randomly ordered four LFSRs and put them into a four-variable Boolean function, and another that selected only three out of four LFSRs to use as inputs in a three-variable function. Our tests on the resulting output strings show a drastic increase in complexity, while simultaneously passing the stringent randomness tests required by the National Institute of Standards and Technology for pseudorandom numbers.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ASCII	American Standard Code for Information Interchange
LFSR	linear feedback shift register
NIST	National Institute of Standards and Technology
NPS	Naval Postgraduate School
PRBG	pseudorandom bit generation
PRNG	pseudorandom number generator
STS	Statistical Test Suite

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

Cryptography is the method by which modern communications become secure. One type of cryptosystem that is moderately secure and quick to implement is the stream cipher, which depends on the ability to create long strings of bits in a sufficiently random method. The random string is then combined with the binary data through the bitwise XOR operation, producing a ciphertext string that seems random itself. After the seemingly unintelligible message has been transmitted, the receiver uses the bitwise XOR operation to combine the message with the same random string used to encrypt the message, leaving behind the original message.

The cryptographic strength of the stream cipher depends primarily on the random string used to encrypt and decrypt the message. The most common method of creating such a bit string is by first creating strings, using a linear feedback shift register, that are not random enough to be secure on their own, and then mixing them in a cryptographically secure manner using a Boolean function called a combiner. Traditionally, this combiner takes the first bit of each input string and assigns each to a variable in the function, then the output of the function becomes the first bit of the output string. The combiner then moves on to the second bits of the input strings, assigns them each to their respective variables, and uses the output of the function as the second bit of the output string. The combiner carries on in the same manner until enough bits have been created to encode the message. Many of the qualities of classical combiners have been studied, and upper bounds on the complexity of an output string based on the complexities of the input strings have been proved.

This thesis introduces a modification to the classical combiner. Instead of assigning the bits from the first input sequence to the first variable, the bits from the second input sequence to the second variable, and so on, we apply a probabilistic approach that randomizes the order of the input sequences such that the order changes at each bit in the sequence. We applied this in two fundamental methods. First, we created four input strings and randomly chose and ordered three of them at each bit to be used for a three-variable combiner. Second, we created four input strings and randomly ordered them at each bit to be used in a four-variable combiner.

We tested the resulting output strings from these probabilistic combiners using multiple

metrics. Primarily, we judged the strings based on their linear complexity, which is a measure of how many bits a linear recurrence relation would have to use in order to describe the sequence perfectly. Our results far exceeded the known bounds for the classical combiner, approaching an infinite linear complexity. We also tested the strings using the National Institute of Standards and Technology's Statistical Test Suite, which is a battery of 15 tests used to judge the randomness of a bit string. Our strings passed almost all of the tests in the suite, with a single explainable failure.

In summary, our new method of bit generation is based on the idea of a combiner as used in stream ciphers, and adds a probabilistic component that significantly increases the linear complexity and statistical randomness qualities of the resulting output strings.

Acknowledgments

I would like to thank my wife, Allison, for all her support through the many months spent working on this thesis. She graciously took care of our newborn daughter while I spent so much time working on this project. I would also like to thank my advisor, Dr. Pantelimon Stănică, and my second reader, Commander Thor Martinsen, for their guidance and help in creating this finished product. Finally, I would like to thank the Applied Mathematics Department at NPS for the education and personal and professional guidance I have received in the last 18 months while working on this degree.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

People in today's world interact using cryptography on a daily basis, although most do not even think about the security it brings or how it works. Following the proliferation of digital communications, including satellite television, smart phones, and the internet, people are sending and receiving massive amounts of data through digital means. Some data, like funny cat pictures, has no need to be sent in a clandestine manner. Other data, however, must be sent with some sort of secrecy or assurance of privacy. When individuals make online purchases, or check online health data from their doctor's office, they expect that their personal financial and health information is kept secret. Similarly, when media companies offer a subscription-only service like streaming video, they must ensure that their data can be viewed only by people who have purchased the service, and thus, must obscure their data before posting it online. This method of keeping data secret is called cryptography, and the process of obscuring the data is called encryption, while the process of unscrambling that data after transmission is called decryption.

1.1 Motivation

Perhaps surprisingly, the security of modern encryption schemes (called cryptosystems) is based not on the obscurity of the mechanism used to encrypt the data, but the strength of the secret key. The key is a string of numbers that tells the scheme how exactly to scramble the data in a particular instance. The creation of a good key, therefore, requires some way to access random numbers, since a nonrandom key would be easy to predict and an adversary could use the known mechanism and guessed key to unscramble, or decrypt, the message. Since computers are deterministic machines, it is impossible to generate truly random numbers in a necessarily deterministic computer program. There are some methods to harvest true random numbers from a computer, like through observing timing information on the hard disk read/write head, but those methods are too slow to generate enough digits to be useful for cryptographic purposes. For this reason, we use programs designed to approximate randomness. These programs are called pseudorandom number generators. Since most data is transmitted in a binary representation, we also use the phrase

pseudorandom bit generator to mean effectively the same tool, but one which generates only ones and zeros rather than the full spectrum of real random numbers.

1.1.1 Lightweight Ciphers

When encrypting data, the parties communicating have many choices of encryption schemes. They must decide not only on the strength of the cryptosystem, but also on how fast the system must work. Understandably, a stronger cryptosystem may take many more operations, and thus more time, than a weaker cryptosystem. In those cases where speed is more important than cipher strength, a party may use what is called a lightweight cipher. This is a cryptosystem that is quick to encrypt, quick to decrypt, and strong enough for the data being transmitted. One application for a lightweight cipher is in cell phone communications. The data transmitted is usually not very sensitive, but the parties speaking would still like some privacy in their communication. A cell phone conversation, however, must be encrypted and decrypted very quickly, since any delay due to encryption would cause confusion and frustration in the call.

One type of lightweight cipher is the stream cipher. Simply stated, the stream cipher generates a "stream," or string, of pseudorandom bits, and bitwise adds them (using the "exclusive or" operation, or XOR) to the data being transmitted to encrypt. The receiver then needs just to generate the same string of bits and XOR them with the encrypted data to decrypt and recover the data that was sent. The speed of this method is restricted only by the method of generating the bit strings, since the XOR of bits is one of the simplest operations to implement in hardware or software.

1.2 Background

In this thesis, we focus on a particular type of stream cipher. This stream cipher is implemented by first generating multiple sequences of bits in a fast but insecure method, and then combining them into a single string using some sort of function that makes the output much more difficult to guess than the input strings. The input strings are generated using linear feedback shift registers (LFSRs), which rely on a linear recurrence relation to generate bits in a rapid manner. The quickly generated strings are then combined using another mechanism referred to (some could say unimaginatively) as a combiner. This thesis

investigates not only the classical method of implementing a combiner for multiple LFSRs, but also a probabilistic alteration to the method that increases the complexity, and therefore, also the security, of the final output string.

1.2.1 Linear Feedback Shift Registers

LFSRs are easy to implement in hardware and in software, and are a very fast way of generating long strings of pseudorandom bits. The bits have some good properties, which will be described in detail in Section 2.3. LFSRs have a critical weakness, however, and in short, that weakness is linearity. Since the sequences are generated by a linear recurrence relation, it is relatively easy to determine that relation based on a small sequence of bits pulled from the larger sequence.

1.2.2 Combiners

Since the LFSR sequences are not sufficient in and of themselves, we must alter them in some way to increase their linear complexity. One common method of such an alteration is the combiner, which is a nonlinear function that takes bits from multiple LFSRs as input variables, and generates a single output string with higher linear complexity than any of the input strings. Traditionally, this combiner function is implemented by labeling each of the input LFSRs (e.g., L_1, L_2), then labeling each of the variables in the function (e.g., x_1, x_2, x_3), and thus the bits from each input LFSR are used as the input in the location of the corresponding variable. This implementation is simple and has some very beneficial and easily proved properties that will be covered in Section 2.5.

Probabilistic Alteration

Our contribution to this area of cryptography is an implementation of an intriguing modification on the classical combiner. Instead of the classical method of applying a combiner, which takes each LFSR and puts it in the same spot in the combiner at each bit, prompted by [1] we used a probabilistic approach to scramble the order of the LFSRs at each bit. We created a program to implement this new approach, and tested the output strings generated and checked if it had better cryptographic properties. Our method created output strings with linear complexities far beyond the known bounds for classical combiners. We further tested the output strings using the Statistical Test Suite published by the National Institute

of Standards and Technology (NIST), and the output strings passed almost all of the tests for pseudorandom bit generators.

CHAPTER 2:
Motivating Examples

Chapter 2 builds the foundation of mathematical background knowledge discussed in Chapter 1. It presents examples of binary representation of characters, binary mathematical operations, and the fundamentals of pseudorandom bit generation (PRBG) using LFSRs and combiners. Furthermore, it discusses some concepts regarding assessing and comparing strings to determine how to compare output strings from the probabilistic approach to the combiner.

2.1 Binary Representation of Characters

Computers store data in a binary representation, or in other words, as ones and zeros. These ones and zeros are known as binary digits, or bits. This is the simplest way to store data in physical media, as well as the easiest way to transfer data using physical means like electrical impulses. Storing data in a binary representation requires a sort of dictionary that converts characters into a unique string of bits. One simple example of such a dictionary is the American Standard Code for Information Interchange (ASCII), as shown in Figure 2.1.

Using such a dictionary, we can convert a message into a series of bits called a bit string. For example, the message "hello" is converted into the bit string 01101000 01100101 01101100 01101100 01101111.

2.2 Binary Addition

Now that we can represent data as ones and zeros, the role of cryptography is to obscure such a string of ones and zeros for transmission over an unsecured line, and then revert the obscured string to the original message once it has been received. This commonly involves performing some sort of mathematical operations using just ones and zeros. For this reason, as it is customary, we introduce a special kind of addition (called the "exclusive or", or XOR) defined only for the integers 0 and 1.

XOR addition is investigated more thoroughly in the branch of mathematics known as field

Figure 2.1: ASCII Code Chart

ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(	0010 1000
N	0100 1110	l	0110 1100	)	0010 1001
				space	0010 0000

This chart shows all binary representations of standard ASCII characters. Source: [2].

theory, which is explained in detail in [3].

XOR addition, denoted with the symbol $\oplus$, works in the same way as traditional addition, with the exception that since one and zero are the only elements of the field, one plus one cannot be two. For our purposes, two is equivalent to zero, so $1 \oplus 1 = 0$. Table 2.1 gives a full account of XOR over the binary field.

With this foundation, we can XOR two bitstrings together. The only other caveat we must discuss is that unlike traditional addition, in our application we XOR each bit position independently of others. We call this procedure the bitwise XOR. There is no "carrying" as in traditional addition. We make that convention in order to maintain message length. An example of adding two bitstrings is shown in Figure 2.2.

Table 2.1: XOR Table

$\oplus$	0	1
0	0	1
1	1	0

This table shows the XOR of integers over the binary field.

Figure 2.2: XOR Example

$$\begin{array}{cccccc} & 1 & 0 & 0 & 1 \\ \oplus & 0 & 0 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 \end{array}$$

This example shows the bitwise XOR of two bitstrings.

The One-Time Pad

One immediate application of bitwise XOR to cryptography is observing that the bitwise XOR can scramble a message. Simply by taking the bits of a message, and combining them using bitwise XOR with a random string of ones and zeros of the same length as the message, we can transform an intelligible message into a garbled string of nonsense. Figure 2.3 is an example of encrypting our example message using bitwise XOR.

Figure 2.3: Simple XOR Encryption

	h	e	l	l	o
	01101000	01100101	01101100	01101100	01101111
$\oplus$	01010110	01111010	11100001	10101110	10000110
	00111110	00011111	10001101	01000010	11101001

Decryption of a message that has been encrypted using bitwise XOR is quite simple. Since each bit is its own additive inverse, decryption requires only that the receiver add the same random string of ones and zeros to the encrypted message in order to decrypt it and recover the original message. The decryption of the message we just created is demonstrated in Figure 2.4.

Figure 2.4: Simple XOR Decryption

	00111110	00011111	10001101	01000010	11101001
$\oplus$	01010110	01111010	11100001	10101110	10000110
	01101000	01100101	01101100	01101100	01101111
	h	e	l	l	o

With some other considerations, we can create an incredibly secure cryptosystem. First, we must require that the key (in this case, the random string of ones and zeros) must be truly random. If the key is not truly random, there may be some way to guess future bits of the key given previous bits. Second, the key must be at least as long as the message being transmitted. If the key were any shorter, we would have to repeat parts of the key in order to have enough bits to XOR with the bits in our message. Third, the key must only ever be used one time, since if we use the same key more than one time, someone who had access to the message the first time we used it would be able to decrypt the second message as well. With these three caveats, simple XOR encryption becomes the cryptosystem known as the One-Time Pad.

The One-Time Pad is, in fact, the only type of encryption that is provably secure against every possible attack [4]. Given any amount of prior information, an attacker can never know what message follows the information he has. Since the key is truly random, there is no possible way to guess the next bit given any number of previous bits. Since the key is the same length as the message, it never repeats, so no previous bits give even a hint as to the bits that follow. Since the key has never been used before, and will never be used again, no amount of knowledge of other encrypted messages can help the attacker crack the message in question. It is perfectly secure against attack.

The One-Time Pad, however, has significant shortfalls when considering applicability. First, it requires a truly random bitstring that is at least as long as the message. Truly random numbers are computationally very inefficient to create (since computers are deterministic machines, any bitstring created strictly by a computer program is inherently not truly random). Generating truly random numbers typically requires measuring natural processes that are considered to be truly random, and techniques that do that are typically very slow [5]. Furthermore, since the key can only ever be used once, a new key must be generated each

time a new message is to be transmitted. These keys must be transmitted and stored as well, so the parties communicating must also have a secure transmission and storage method in place to safeguard the key until it is to be used. Therefore, although the One-Time Pad is provably perfectly secure, it is impractical to implement in most cases.

2.3 Linear Feedback Shift Registers

Since computers require longer strings of bits than can be generated in a reasonable amount of time by truly random number generators, we must find a way to generate a pseudorandom sequence of bits quickly. In order to be cryptographically secure, however, we must also add the constraint that this string has certain properties. Therefore, before explaining how to generate strings with good properties, it is important to explain what properties make a bitstring good.

In order to be considered random enough for cryptographic purposes, a bit string must have the property that given any number of bits from the string, an attacker should have no greater than a .5 chance of guessing the next bit (that is, there is no bias in the generated bits). The first property that comes from this requirement is that the string must be balanced, which means it must have approximately the same amount of ones as it has zeros. Clearly, if a string was not balanced, an attacker could simply guess the more frequently occurring bit and be correct more than 50 percent of the time.

The next property that is important for a pseudorandom bit sequence is that it has a good distribution of runs. A run is a repeated substring of the same bit, like 00000 or 111111111. A good bit string must have the right amount of runs of the right lengths. Clearly, a string with too large a proportion of long runs would make it easier to guess the next bit, since it would be more likely to be the same bit as the preceding bit. Conversely, too high a proportion of very short runs is also a weakness, since an attacker could simply guess that the next bit will be opposite the last bit. Mathematically speaking, a good distribution of runs means that the frequency of occurrence of a run of length one is $\frac{1}{2}$, the frequency of a run of length two is $\frac{1}{4}$, and so on. In general, the frequency of occurrence of a run of length n should be $\frac{1}{2^n}$.

Finally, a strong bit string should not repeat in a short period. If an attacker can recognize

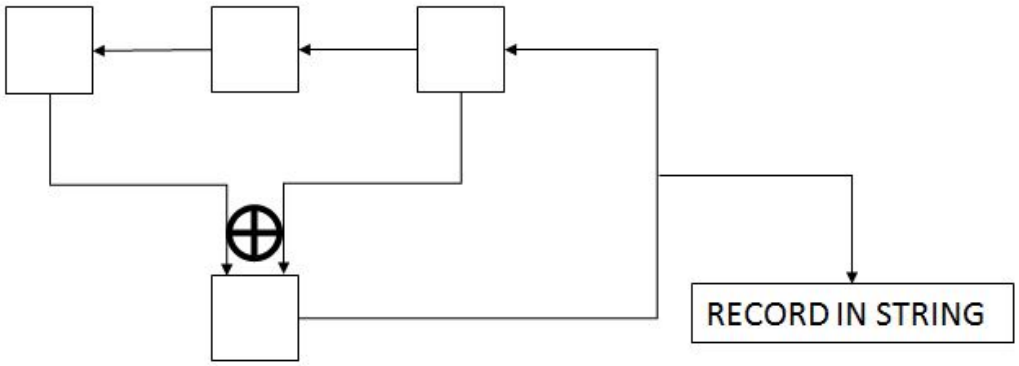
a period of repetition, then he can simply look back one period into the past and guess the next bit with 100 percent accuracy.

Our goal, then, is to create an algorithm that quickly generates a long string of bits that satisfies those three properties, as well as some others specified by the NIST [6]. An LFSR is one way to generate a long string of bits that are balanced and have a good distribution of runs. It works by introducing a recurrence relation such that the next bit in a sequence is defined by some function of the previous bits. One such equation is

$$x_{n+3} = x_n \oplus x_{n+1}. \tag{2.1}$$

Implementing an LFSR like the one in Equation 2.1 is simple, whether in hardware or software. One way to think of how the LFSR works is referred to as the Fibonacci register. Since our example recurrence relation looks back three bits, we set up our register with three blocks of memory. Then, we mark the positions that will be added together with a line, called a tap. The empty register is shown in Figure 2.5.

Figure 2.5: Empty Fibonacci Register



The next step is to fill in the blocks with a binary seed, or initial value, chosen at random. We use the sequence 110 as our initial value. Once the blocks are filled, we XOR the values that are tapped, shift each value to the next block, and record the new value as the first bit in our output string. This process is shown in Figures 2.6 and 2.7.

Figure 2.6: Filled Fibonacci Register

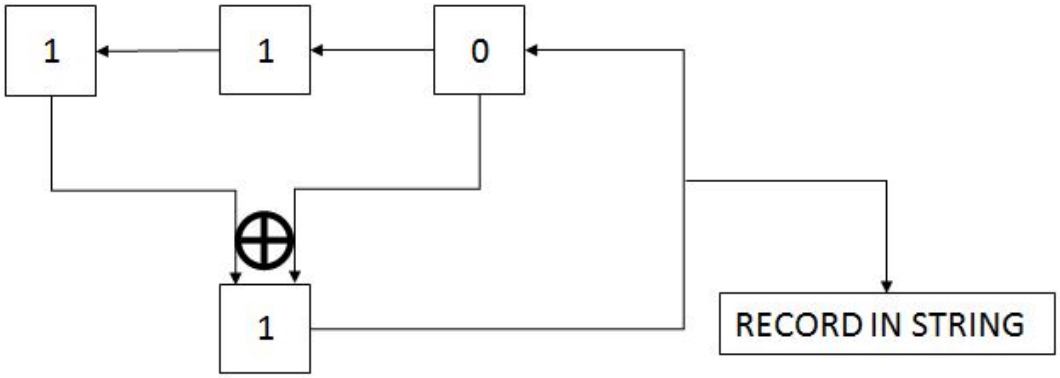
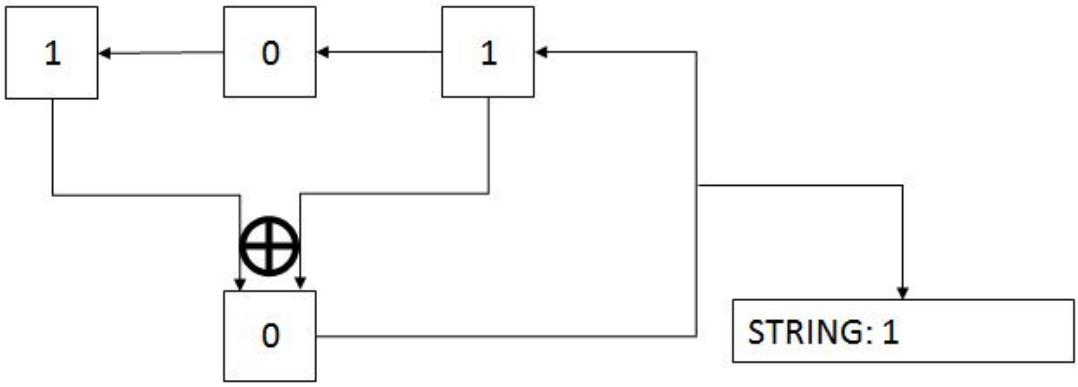


Figure 2.7: Shifted Fibonacci Register



It is important to note that the initial values cannot all be 0, since when we add them together later they will always produce a 0, and thus, the final bit string will be a string of 0s. Clearly, setting all of the initial values to 0 does not produce a very random bit string.

We have now created a bit string using a very simple method. Furthermore, this string is guaranteed to be balanced and have an optimal distribution of runs. What is more, with careful choice of a recurrence relation, we can also guarantee a maximum period of $2^n - 1$, where n = number of bits in the recurrence, before the sequence repeats. This maximal period sequence, or m-sequence, occurs because there are 2^n possible states for the register,

and the best sequence achieves all of the states save the state of all zeros, as discussed in the previous paragraph. This yields a maximal period of $2^n - 1$. Further discussion of the choice of the involved recurrence relation can be found in [7].

2.3.1 Linear Complexity and the Berlekamp-Massey Algorithm

The strength of a sequence generated by an LFSR is given by the minimum number of bits in the recurrence relation that would generate that sequence. For our example sequence, the recurrence looked back three bits, and there is no shorter recurrence that would generate the same sequence. Therefore, we say that the sequence has a linear complexity of 3. A sequence that required a recurrence relation that looked back 12 bits would have a linear complexity of 12.

The main weakness of an LFSR is due to the fact that, although it has good balance, a good distribution of runs, and good length before repeating, the relationship is linear. This fact makes it relatively easy to guess the recurrence relation given only a portion of the sequence. In fact, given only $2L$ bits, where L is the linear complexity of a string, the Berlekamp-Massey algorithm is guaranteed to find the recurrence relation that generated that string [8]. Therefore, although an LFSR has many desirable properties, it is insufficiently secure for cryptographic purposes.

2.4 Binary Multiplication

Since an LFSR only uses the XOR operation, and it is insufficient to generate a sequence with high enough complexity, we introduce binary multiplication. Multiplication over the binary field works exactly the same as traditional multiplication. Since zero and one are the only elements of the field, and every product of the two of them remains within the field, no changes or clarification is needed to describe multiplication. Table 2.2 shows all products over the binary field.

Similar to binary addition, we multiply corresponding bits in a bitwise manner in order to maintain message length.

Unfortunately, binary multiplication alone is insufficient to create a cryptosystem like the One-Time Pad, because there would be no way to decrypt the encoded message. A 0 in the

Table 2.2: Binary Multiplication

·	0	1
0	0	0
1	0	1

This table shows all the products of integers over the binary field.

Figure 2.8: Bitwise Binary Multiplication Example

$$\begin{array}{cccccc} & 1 & 0 & 0 & 1 \\ \cdot & 0 & 0 & 1 & 1 \\ \hline & 0 & 0 & 0 & 1 \end{array}$$

encrypted message, when paired with another 0 in the key location, would leave ambiguity with respect to the plaintext, since either a 0 or 1 could have produced the same resulting ciphertext. Regardless, multiplication provides the method to break the linearity of a string created from an LFSR.

2.5 Combiners

Now that we have binary addition as well as multiplication, we can combine multiple LFSR strings with a nonlinear function in order to create a more complex string that is more resistant to analysis via the Berlekamp-Massey algorithm. We will discuss combiners on three LFSR strings, though the concept can be extended to any number of strings.

A classical combiner is simply a Boolean function that takes some number of LFSR-generated strings as arguments, and produces another bit string. The least complex combiner would be simply adding together each of the bit strings, using the equation

$$f_1(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3. \tag{2.2}$$

Using some LFSR bit strings as an example, this combiner would generate a new sequence, as shown in Figure 2.9.

Figure 2.9: Linear Combiner Example

x_1	1101001110100111
x_2	1010110010001111
x_3	1100110100100001
f_1	1011001000001001

While the string produced by this combiner does have increased complexity over each of the input strings, the increase is bounded by the periodicity of each of the input strings, and so the complexity is limited by the sum of the complexities of the input strings. In order to get a greater increase in linear complexity, we must introduce a nonlinear component to the combiner. By using nonlinear Boolean functions in the combiner, we disguise the linearity of the resulting string, and make it much harder to analyze. One example of a nonlinear combiner is

$$f_2(x_1, x_2, x_3) = x_1 \oplus x_1x_2 \oplus x_3. \tag{2.3}$$

Using the same input strings, we come up with a completely different output string, as demonstrated in Figure 2.10.

Figure 2.10: Nonlinear Combiner Example

x_1	1101001110100111
x_2	1010110010001111
x_3	1100110100100001
f_2	1001111000000001

Since there is a nonlinear component in the combiner from Figure 2.10, the complexity increases significantly more than in a linear combiner.

2.5.1 Bounds on Linear Complexity

Upper bounds on linear complexity for classical combiners have been studied extensively. Most importantly, the linear complexity of an output string, given input LFSR strings X_1 through X_n, with corresponding complexities L_1 through L_n, after being combined in an n-variable function $f(X_1, ..., X_n)$, is bounded by $f(L_1, ..., L_n)$ [9]. Furthermore, if the linear

complexities of the input LFSR strings are coprime, then we are guaranteed an output string with the maximum possible linear complexity.

In our example, we used input strings with linear complexities 3, 4, and 5, respectively. Therefore, using a linear combiner, the best possible complexity of an output string we could achieve is $3 + 4 + 5 = 12$. The Berlekamp-Massey algorithm would still be able to generate a recurrence relation quickly on that string. When we introduced a nonlinear combiner in the second example, however, the upper bound for complexity is $3 + (3)(4) + 5 = 20$, which is much greater than the linear combiner. In this small example, the algorithm would still be able to return a recurrence relation, but it would be more difficult. Furthermore, as the number and complexities of the input strings are increased, as well as the degree of the combiner, this increase in linear complexity becomes so strong that the Berlekamp-Massey algorithm becomes computationally impractical in attacking the string.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Methodology

Chapter 3 describes the rationale behind the implementation of the probabilistic combiner. It also discusses specifics regarding the method of implementation, beginning with constructing an LFSR, and continuing through the classical and then probabilistic alteration to a combiner. All examples of code created are available in the Appendix.

3.1 Language

We chose to write the code using R, which is a statistical programming language built using mostly the C language. We chose to write in R primarily due to familiarity with the language, but also because the language lends itself nicely to probabilistic applications and has many built-in functions for random number generation and random sampling. Though the code used for randomness in R may not be cryptographically secure, it suffices for our purposes, since the security in our code does not rely on the random sampling, but rather on the implementation of the combiner.

3.2 Coding Strategy

We decided to write this code from scratch, instead of using preexisting code, for a variety of reasons. Primarily, we wanted to be able to build the code in such a way that it was highly adaptable to multiple applications later. For example, when coding the LFSR portion of the program, we allowed the output length and recurrence relation to be independent input variables, so that changing the recurrence relation did not require a large amount of effort.

3.3 Classical Combiner

In order to code the probabilistic combiner, we first coded a standard combiner so that we could verify that it functioned properly, then added the probabilistic modification. We wrote a program that created a single LFSR, with some careful planning involved. In order to accommodate adjustments later, we wrote the program to accept output string length and recurrence relation as variables, rather than hard-coding them into the LFSR. Because of

this, the code easily accepts any recurrence relation, should a user want to use a specific LFSR string in their analysis. Also, since testing required analysis of bit strings of varying length, having a single variable to accommodate output length made this analysis much easier. Furthermore, since each LFSR requires some number of seed values, we created code to generate each seed value as a random sample from a uniform distribution on [0, 1], rounded to the nearest integer. We calculate the length of the seed based on the length of the recurrence relation. As an important note, we did not code a failsafe to ensure the seed was not the string consisting of all 0's. Since the seed was generated from a uniform distribution, however, there is only a chance of $\frac{1}{2^n}$ of receiving that string, where n = the number of bits in the seed. All of our recurrence relations were of size at least 8, which means the probability of generating the string of all 0's is no greater than .004. What is more, even if such a string were generated, it would be obscured after applying the combiner process. We considered this an acceptable risk, though anyone who desired to add such a failsafe could do so with little difficulty by adding code to check if the seed was the 0 string, and making the code repeat in case it was.

After creating the code to build a single LFSR, we scaled it up to build multiple LFSRs and store the result in a two-dimensional array. The array allowed for easy lookup during the combiner phase of the program, since each LFSR could be called from the same object.

Coding the combiner involved simply calling each LFSR from the LFSR array, and adding or multiplying as called for by the combiner function. In our program, we chose simple combiners of low degree that were balanced. This allowed for easy comparison between the classical and probabilistic combiners, because the properties and bounds were easy to calculate in the classical case. For a more detailed discussion on Boolean functions and their properties, see [10].

When programming the combiner, we used a construct that allowed for easy conversion to the probabilistic combiner. Instead of simply coding the combiner function to look up specific rows corresponding to specific LFSRs in the LFSR array, we defined a constant vector of the same size as the combiner, and set the combiner to call the row from the LFSR array corresponding to the particular entry in the vector. Finally, we programmed the classical combiner to save its output in a separate array, so that it could be exported simply.

3.4 Probabilistic Combiner

Once the classical combiner was working properly, we implemented the probabilistic component. Simply, we added a line of code that redefined the constant vector described above as a random permutation of the integers from 1 to n, where n = the number of LFSRs in the array. That line was placed inside the loop that repeated for each bit in the output string, so the combiner read a randomized permutation of LFSRs before calculating the function at each bit. For example, when calculating one bit, the LFSRs might be fed into the combiner function in the order $(1, 3, 4, 2)$, but the next bit might be calculated with the order $(4, 1, 2, 3)$. By making this one small change to a single line of code, we completely changed the combiner process. We implemented the change in a few different applications.

3.4.1 4 Choose 3

One application of the probabilistic combiner was generating more LFSRs than the combiner function needed to accept. Then, by generating random permutations of integers as described above, but of a size equal to the size of the combiner, we can not only introduce the complexity of more LFSRs than would normally be included in such a combiner, but also increase complexity by randomizing which LFSR is used to calculate each bit. We used the combiner in Equation 2.3.

Though the complexity increase is drastic using this method, we wanted to make sure that we were comparing resulting bit strings in as fair a manner as possible. Using our method, we added complexity in two different ways: first, by combining four LFSRs in a three-combiner, and second, by randomizing the order in which each of those LFSRs were added. We felt it unfair to compare the classic combiner's performance to this new combiner, because adding the complexities of four LFSRs into a three-combiner would clearly increase the complexity, when what we wanted to test was primarily the effect of the randomization. This led us to the 4 choose 4 combiner, which is described below.

Linear Combiner

One interesting application of a probabilistic combiner that uses more LFSRs than the combiner requires is the linear combiner. As discussed in Section 2.5, a linear function is a terrible choice for a combiner, since it increases the linear complexity by such a small amount. When a probabilistic approach like ours is applied, however, the complexity

increase is much more difficult to predict, since we introduce more randomness by choosing which LFSRs are included when calculating each bit.

3.4.2 4 Choose 4

In order to isolate the effect of randomizing the order of the LFSRs at each bit from the effect of including more LFSRs than a combiner would normally accept, we decided to implement the probabilistic combiner on a function that accepts the same amount of input strings as we generated. This approach enables us to compare the probabilistic combiner with the traditional combiner in a more fair manner. We chose to use the function

$$f_3 = x_1 \oplus x_2x_3 \oplus x_4. \tag{3.1}$$

This equation is not the greatest choice for a realistic combiner application, as it has nonlinearity 4, but it is balanced, and it has correlation immunity 1, but was sufficient for our purposes. See [10] for more details on these cryptographic concepts).

3.5 Analysis Tools

The primary measure of randomness that we compared our results with was linear complexity, as explained in Section 2.3. In order to calculate the linear complexity of our output strings, we exported the strings to be analyzed in a different program than R, since there were no convenient analysis tools built into R. We primarily used Wolfram Mathematica, and code that was created by Galbreath and extended by Gerhardt [11]. The code runs the Berlekamp-Massey algorithm on a bit string and outputs the linear complexity as well as the linear complexity profile, and the corresponding recurrence relation. The linear complexity profile is a way to track the linear complexity increases as the algorithm runs; it also produces an easy way to visualize the results in a graph. A good bit string increases complexity at a rate approximately equal to $\frac{n}{2}$, where n is the number of bits analyzed up to that point in the algorithm. Unfortunately, the algorithm implementation in Mathematica could only analyze bits strings of length up to $10,000$ bits before it became too slow to run.

We also used a battery of tests prescribed by the NIST to judge the randomness of bit strings. These statistical tests, collectively referred to as the NIST Statistical Test Suite (STS),

give results as p-values, where the null hypothesis is randomness. Thus, a low p-value would imply nonrandomness, and a p-value that fails to reject the null hypothesis implies sufficient randomness. These tests were also implemented using Wolfram Mathematica, and code written by Galbreath and Gerhardt [12]. Similar to the linear complexity tests in Mathematica, though, the STS code ran too slowly on bit strings that were too long. For this reason, we also used an implementation of the STS written in Python and C by Justamante [13] to test strings up to two million bits in length.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results

Chapter 4 details the results of various tests on the output strings created by both the classical combiner, as well as both the 4 Choose 3 and the 4 Choose 4 probabilistic modifications. We will first discuss results of linear complexity tests, and compare the theoretical bounds described in Section 2.5 with the actual results we obtained using each combiner. Finally, we will show the performance of the strings when subjected to the NIST Statistical Test Suite.

4.1 Linear Complexity

The primary measure of randomness that we used to judge the performance of our various combiners was linear complexity. The reason we chose linear complexity is because that is the most common measure of randomness used when analyzing LFSRs and combiners. Unfortunately, there was no easily accessible test for linear complexity in R. Therefore, we implemented the test in a different programming language: Wolfram Mathematica. The test in Mathematica provided as outputs not just the linear complexity, but also the recurrence relation and the linear complexity profile. Unfortunately, it ran too slowly to accept bit strings longer than about $50,000$ bits. That said, we feel that the insight gained from testing the shorter bit strings is sufficient for our purposes.

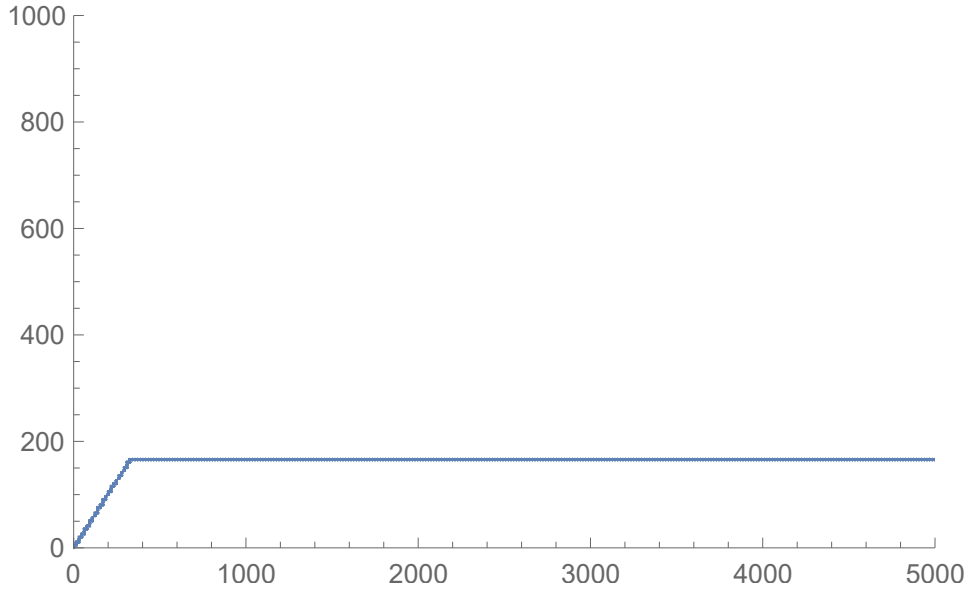
One important note to remember is that the Mathematica program relies on the Berlekamp-Massey algorithm to calculate linear complexity. As mentioned in Section 2.3, the algorithm requires $2L$ bits to determine a linear complexity of L. Thus, the practical limit for any linear complexity test is half of the bits of the input string. For example, a truly random bit string of length $100,000$ bits would return a linear complexity of $50,000$ when put through the Berlekamp-Massey algorithm.

4.1.1 Classic 3-Combiner

Before testing the probabilistic combiner, we ran the tests on output strings of the classical combiner for comparison. The input LFSRs that we used for all of the tests had complexities

$L_1 = 8$, $L_2 = 9$, $L_3 = 11$, and $L_4 = 13$. We chose those complexities because they are coprime and thus ensure that we achieve the upper complexity bounds when combined via the classical combiner. The function we used for the 4 Choose 3 combiner was the same as Equation 2.3. In order to maximize the linear complexity of the output string, we chose to use the fourth LFSR for x_1, the third as x_2, and the second as x_3. By the formula presented in Section 2.5, the expected complexity of the output bit string was $13 + (13)(11) + 9 = 165$. In fact, this is exactly the result we achieved. Figure 4.1 shows the linear complexity profile of the output string. As shown in the figure, the linear complexity increased at a steady rate until 330 bits had been processed, at which point the complexity leveled off at 165. This shows the point where the algorithm had determined a minimal recurrence relation that describes the sequence, and in this case, it took 165 bits to describe a relation for the new bit string.

Figure 4.1: Linear Complexity Profile of the Classical 3-Combiner

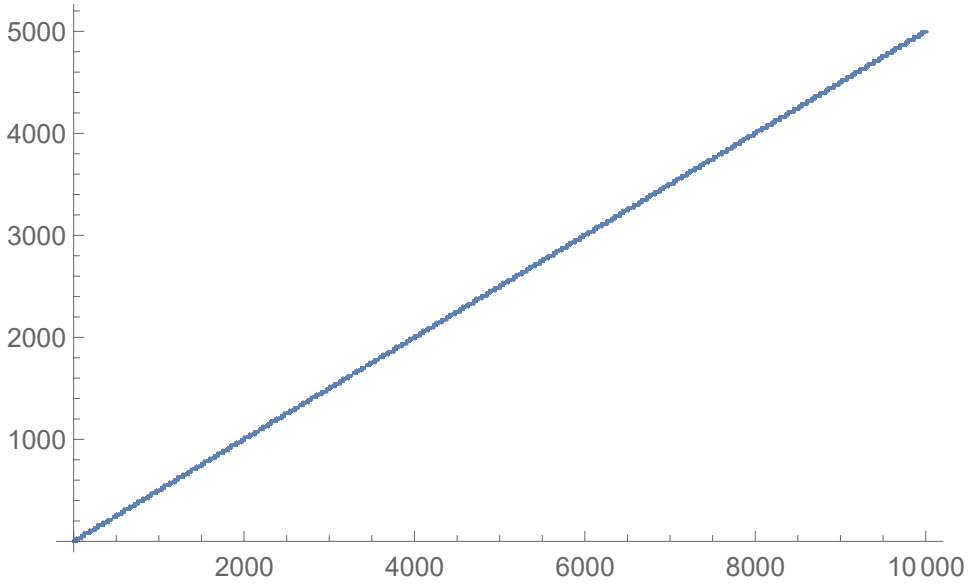


4.1.2 Probabilistic 4 Choose 3 Combiner

We then analyzed the probabilistic combiner, using the same combiner function and all 4 LFSRs. When analyzing a string of $10,000$ bits, we received a linear complexity of $5,001$, meaning the algorithm continued to run until it ran out of bits to analyze, and still had not

found a valid recurrence relation. The linear complexity profile, as shown in Figure 4.2, continued to grow in complexity as the algorithm ran and never leveled off.

Figure 4.2: Linear Complexity Profile of all Probabilistic Combiners



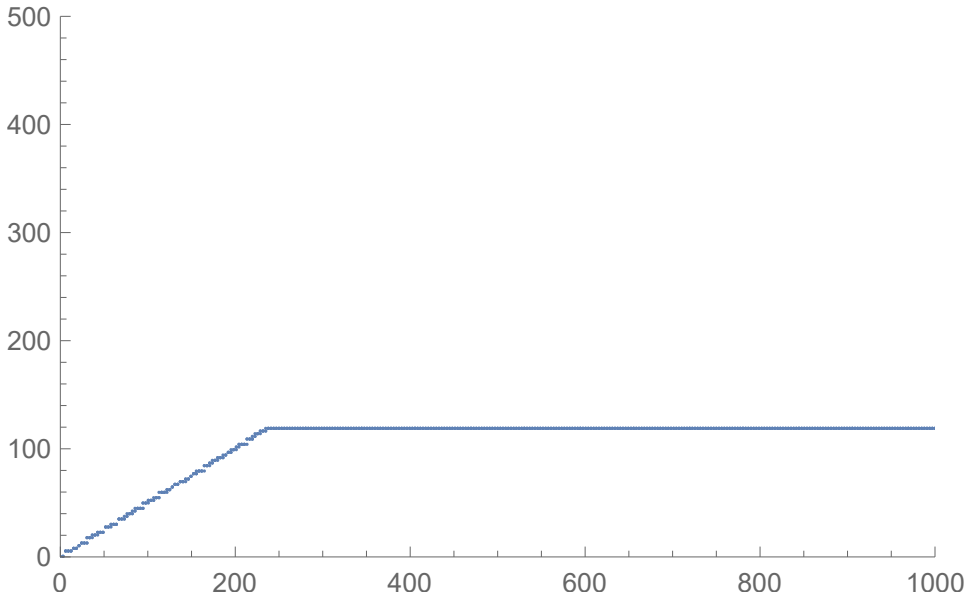
We further analyzed the 4 Choose 3 probabilistic combiner by changing the combiner function to the linear combiner. Clearly, the best possible complexity using the same input LFSRs and the classical combiner would be 13 + 11 + 9 = 33, but we wanted to see if the probabilistic method could exceed that bound. In fact, the probabilistic method drastically exceeded the bound. Much like when using the nonlinear combiner above, output strings from the probabilistic combiner showed linear complexity profiles that continued to increase as the strings we analyzed got longer and longer. Its linear complexity profile was identical to the one shown in Figure 4.2. As the algorithm continued to run, the complexity of the string increased at a steady rate, showing no sign of leveling off.

4.1.3 Classic 4-Combiner

We continued our tests by looking at the classical 4-combiner. As discussed in Section 3.4, we wanted to isolate the effects of the probabilistic randomization from the effects of having too many input LFSRs for the combiner. To get accurate baseline data, we tested the combiner in Equation 3.1, using the same four LFSRs as above. The upper bound for linear

complexity of the output string, then, is $8 + (9)(11) + 13 = 120$. When we tested an output string of $10,000$ bits in the Mathematica program, we saw exactly what we expected. The output string had a linear complexity of 120, and the linear complexity profile leveled out at 120 after 240 bits were analyzed. The linear complexity profile is shown in Figure 4.3

Figure 4.3: Linear Complexity Profile of the Classical 4-Combiner



4.1.4 Probabilistic 4 Choose 4 Combiner

Similar to the results of the 4 Choose 3 probabilistic combiner, the 4 Choose 4 combiner gave an output string with seemingly infinite linear complexity. The $10,000$ bit string gave a linear complexity of $4,999$. Its linear complexity profile was also identical to the one shown in Figure 4.2.

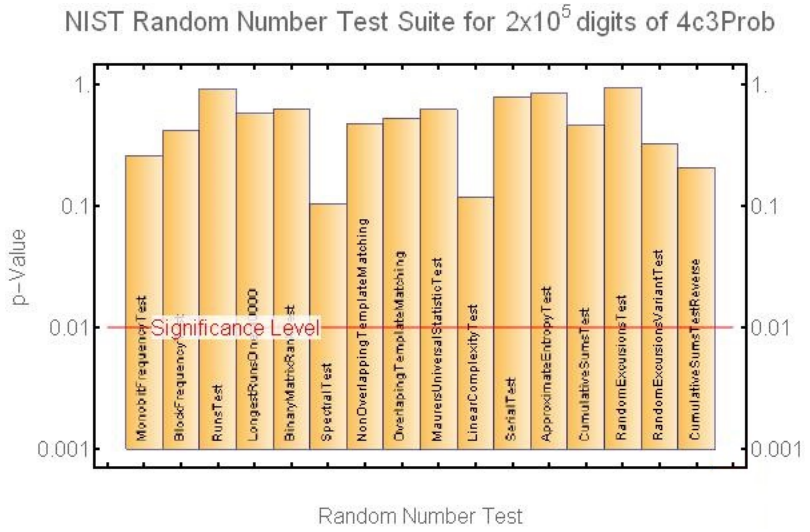
4.2 NIST Statistical Test Suite

We finished evaluating our bit strings by testing the strings using the NIST STS. The STS currently consists of 15 different statistical tests, each designed to evaluate different qualities that a random bit string should have. The simplest tests evaluate balance and distribution of runs, and the others test more complex randomness qualities. One important note is that the code we used was written in 2010, and also in 2010 the NIST removed two tests (namely

the Lempel-Ziv Compression Test and the Cumulative Sums Reverse Test) from the suite. Our results include the Cumulative Sums Reverse Test, but it is no longer part of the official STS.

We tested each of the three types of probabilistic combiners using the STS. Initially, we used a program written by [12] in Mathematica that produced easy to read charts with the results. We were able to test up to 200,000 bits before the program ran too slowly to be useful. Those results are shown in Figures 4.4, 4.5, and 4.6.

Figure 4.4: STS Results for 200K Bits from the Probabilistic 4 Choose 3 Combiner



In order to test longer strings (over 1,000,000 bits), we used an implementation of the STS created by [13], written in C and Python. We tested strings of length 2,000,000 bits from each of our three implementations of the probabilistic combiner. Table 4.1 shows the results of the tests on the longer bit strings, and Table 4.2 shows the order in which the tests were performed.

Clearly, strings generated by our probabilistic combiner have a very strong passing rate for the tests required by the NIST. The only test that failed (p-value below $\alpha = .01$) was the Discrete Fourier Transform test on the 4 Choose 3 nonlinear combiner. We expected this test to fail, since the combiner function we used is not a particularly strong function, partially due to the fact that we were using so few LFSRs, as well as the fact that we were

Figure 4.5: STS Results for 200K Bits from the Probabilistic 4 Choose 3 Linear Combiner

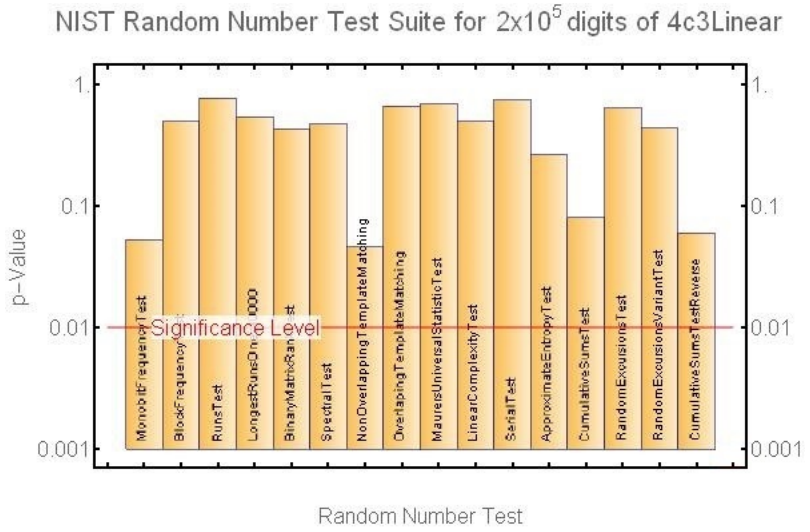


Table 4.1: NIST STS Results

String/Test	1	2	3	4	5	6	7	8
4C3(linear)	.392	.080	.612	.691	.562	.123	.377	.559
4C3	.886	.680	.442	.242	.948	.000	.672	.424
4C4	.319	.517	.525	.465	.831	.726	.757	.357
String/Test	9	10	11	12	13	14	15	
4C3(linear)	.167	.287	.671	.284	.575	.531	.891	
4C3	.306	.660	.525	.353	.533	.349	.936	
4C4	.458	.225	.975	.383	.976	.051	.956	

This table shows results (p-values) of the STS on our bit strings

only using a three-variable combiner. Any realistic implementation of a combiner would use many more LFSRs, and a combiner function with far more variables. The high pass rate implies that the strings we have generated far surpass the randomness generated by a classical combiner, and that by simply mixing up the order of the LFSRs at each bit during combiner calculations, we can introduce a sufficient amount of randomness to be cryptographically secure.

Figure 4.6: STS Results for 200K Bits from the Probabilistic 4 Choose 4 Combiner

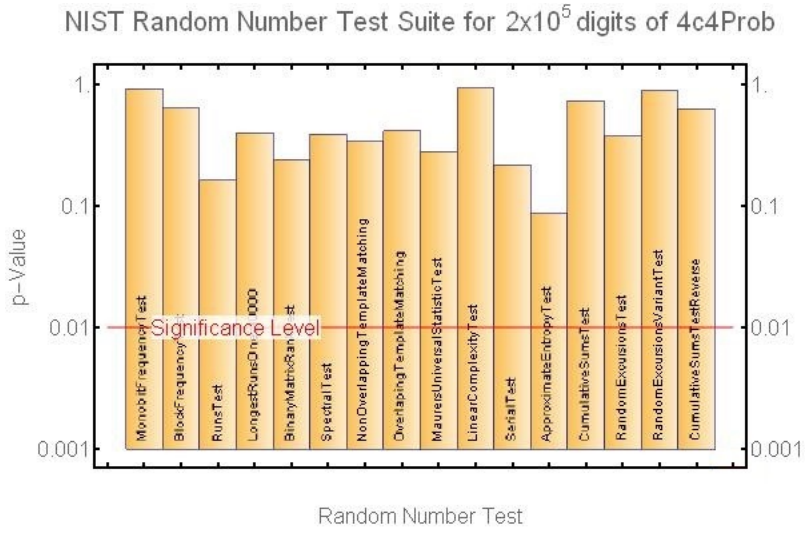


Table 4.2: NIST STS Tests

Test Number	Test
1	Monobit Frequency Test
2	Block Frequency Test
3	Runs Test
4	Longest Runs Test
5	Binary Matrix Rank Test
6	Spectral Test
7	Nonoverlapping Template Matching
8	Overlapping Template Matching
9	Maurer's Universal Statistical Test
10	Linear Complexity Test
11	Serial Test
12	Approximate Entropy Test
13	Cumulative Sums Test
14	Random Excursions Test
15	Random Excursions Variant Test

This table shows the order of the tests in Table 4.1

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion and Future Work

This thesis analyzed the effects of a probabilistic combiner on bit sequences generated by LFSRs. We generated code that modified the classical combiner method to include a probabilistic component, then analyzed the resulting strings with a variety of tests.

The primary metric we used to quantify our bit strings was linear complexity. While the upper bound on the linear complexity of the output string of a classical combiner function is well known [9], we far exceed this bound using the probabilistic combiner. Furthermore, for any string of length n that we analyzed with the Berlekamp-Massey algorithm, we consistently saw a linear complexity of about $\frac{n}{2}$. This result implies that our probabilistic combiner created a functionally infinite complexity bit string (it would be quite interesting if that is the case, in general).

We also subjected our bit strings to the NIST STS, which is the current standard for testing the randomness of bit strings. Our bit strings passed almost all of the tests in the suite, even when the classical combiner using the same input LFSR strings failed the tests.

5.1 Future Work

There is much opportunity for future work using this foundation. Dr. Stănică and CDR Martinsen are currently analyzing the deeper mathematical foundation of the results that we found, to see if anything can be proved regarding the linear complexity of a bit string generated by this method [1].

Comparisons between the performance of the probabilistic combiner, used as a PRBG, against other known PRBGs like Blum-Blum-Shub (described in [14]) or Blum-Micali, could demonstrate interesting qualities of the output strings. If the probabilistic combiner can generate bits more rapidly, and at least as randomly, as these other generators, then the applications could be widespread. Any current application of pseudorandom numbers, cryptographically secure or not, could potentially be expedited and made more secure by this probabilistic modification to the combiner.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Code Examples

The code below was used to create sequences of bits to be tested. All code in this section was written by Nicholas Sharpe, using R.

A.1 LFSR Code

The code below creates a single LFSR. The variable *LFSRlen* is the desired output length for the LFSR, and *outmat* is initialized to be a matrix of zeros large enough to hold each LFSR created. The recurrence relation is stored in the variable *prim*. By repeating the lines from *prim* through the end, we easily created multiple LFSRs, and simply changed the recurrence in each one to ensure they were different.

```
LFSRlen = 200000
outmat = matrix(rep(0,times=4*LFSRlen),4,LFSRlen)

prim = c(1,0,1,1,0,0,0,1,1)
bits = length(prim)-1
seed = c(round(runif(bits),0))
for(i in 1:(bits)) {
  outmat[1,i] = seed[i]
}

for(i in (bits+1):LFSRlen) {
  for(j in 1:(bits)) {
    if(prim[j]==1) {
      outmat[1,i]=(outmat[1,i]+outmat[1,(i-bits+j-1)])%%2
    }
  }
}
```

A.2 Classical Combiner

Once the *outmat* matrix had been created, the code for the combiner simply required inputting values from the matrix into the proper Boolean function, as we did below. We added the *p* vector as a placeholder in the classical combiner that could be modified in the future to allow the probabilistic combiner to work properly.

```
outstrings = matrix(rep(0,times=2*LFSRlen),2,LFSRlen)
p = c(1,2,3,4)
for(i in 1:LFSRlen) {
    outstrings[1,i]=sum(outmat[p[1],i]+outmat[p[2],i]*
        outmat[p[3],i]+outmat[p[4],i])%%2
  }
```

A.3 Probabilistic Combiner

The probabilistic combiner required a single line of code be changed from the classical combiner. Instead of using a constant *p* vector, as in the classical combiner, we moved the line inside the loop, and calculated a new *p* vector for each bit that we put through the combiner.

```
for(i in 1:LFSRlen) {
    p = sample.int(4,size=4)
    outstrings[2,i]=sum(outmat[p[1],i]+outmat[p[2],i]*
        outmat[p[3],i]+outmat[p[4],i])%%2
  }
```

The code for the 4-choose-3 combiner was very similar, and only required a different Boolean function at the end of the code.

List of References

[1] P. Stănică, "A new approach on pseudorandomness: probabilistic combiner," 2016, unpublished manuscript.

[2] R. Parker, "ASCII code: Character to binary," 2007. [Online]. Available: http://rossparker.org/bwa/.

[3] J. B. Fraleigh, *A First Course in Abstract Algebra*. Pearson Education, Inc, 2003.

[4] C. E. Shannon, "Communication theory of secrecy systems," *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[5] M. Haahr, "Random.org: True random number service," 2016. [Online]. Available: https://www.random.org/randomness/

[6] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," Booz-Allen and Hamilton Inc, Mclean, VA, Tech. Rep., 2001.

[7] S. W. Golomb, *Shift Register Sequences*. Aegean Park Press, 1982.

[8] J. Massey, "Shift-register synthesis and bch decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969.

[9] A. Canteaut, *Combination Generator*. Boston, MA: Springer US, 2011, pp. 222–224. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-5906-5_338.

[10] T. W. Cusick and P. Stănică, *Cryptographic Boolean Functions and Applications*. Elsevier – Academic Press, 2009.

[11] N. Galbreath and I. Gerhardt, "lfsr.nb," 2005. [Online]. Available: https://gerhardt.ch/random.php;

[12] I. Gerhardt, "randtest.py," 2010. [Online]. Available: https://gerhardt.ch/random.php.

[13] D. Justamante, "STS implementation," private communication, 2016.

[14] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM Journal on Computing*, vol. 15, no. 2, pp. 364–383, 1986.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California